Access Method Concurrency with Recovery

David Lomet
Digital Equipment Corp.
Cambridge Research Lab
One Kendall Sq., Bldg. 700
Cambridge, MA 02139

Betty Salzberg
College of Computer Science
Northeastern University
Boston, MA. *

Abstract

Providing high concurrency in B$^+$-trees has been studied extensively. But few efforts have been documented for combining concurrency methods with a recovery scheme that preserves well-formed trees across system crashes. We describe an approach for this that works for a class of index trees that is a generalization of the B^{link}-tree. A major feature of our method is that it works with a range of different recovery methods. It achieves this by decomposing structure changes in an index tree into a sequence of atomic actions, each one leaving the tree well-formed and each working on a separate level of the tree. All atomic actions on levels of the tree above the leaf level are independent of database transactions, and so are of short duration.

1 Introduction

The subject of concurrency in B$^+$-trees has a long history [1, 6, 14, 16, 17, 18]. Most work, with the exception of [5, 14], have not treated the problem of system crashes during structure changes. In this paper, we show how to manage **concurrency and recovery** for a wide class of index tree structures, single attribute, multiattribute, and versioned. Further, our approach works for a large class of recovery methods.

The four innovations that make it possible for us to provide high concurrency for index trees are the following:

1. We define a search structure, called a **Π-tree**, that is a generalization of the B^{link}-tree [6]. Our concurrency and recovery method is defined to work with all search structures in this class. Recent work [19] suggests that approaches based on B^{link}-trees should have higher concurrency than other proposed methods. Thus, our technique has both good performance and very broad applicability.

2. Π-tree structure changes consist of a **sequence** of atomic actions [7]. These separate actions are serializable and are guaranteed to have the all or nothing property by the recovery method. Searchers can see the intermediate states of the Π-tree that exist between these atomic actions. Hence, complete structural changes are not serializable. By contrast, in ARIES/IM [14] complete structural changes are *serial.*

3. We define separate actions for performing updates at each level of the tree. Update actions on non-leaf nodes are separate from any transaction whose update triggers a structure change. Only node-splitting at the leaves of a tree may need to be within an updating transaction. Even in this case, only for page-oriented UNDO recovery systems do locks on the split nodes need to be kept to the end of the transaction. This is unlike [14] where entire structure changes are subtransactions of database transactions and where non-page-oriented UNDO recovery must be supported.

4. When a system crash occurs during the sequence of atomic actions that constitutes a complete Π-tree structure change, crash recovery takes no special measures. A crash may cause an intermediate state to persist for some time. The structure change is completed when the intermediate state is detected during **normal** subsequent processing by scheduling a completing atomic action. The state is tested

This work was partially supported by NSF grant IRI-88-15707 and IRI-91-02821

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0351...$1.50

again in the completing atomic action to ensure the idempotence of completion.

The rest of this paper is organized in the following way. Section 2 defines the Π-tree. Section 3 describes the operations on Π-trees. In section 4, atomic actions are described in general. Section 5 describes how Π-tree structure changes are decomposed into atomic actions, and how to cope with such decomposed changes. Finally, section 6 is a short discussion of what we have accomplished.

2 The Π-tree

2.1 Structural Description

Informally, a Π-tree is a balanced tree, and we measure the level of a node by the number of child edges on any path between the node and a leaf node. More precisely, however, a Π-tree is a rooted DAG because, like the B^{link}-tree, nodes have edges to sibling nodes as well as child nodes. All these terms are defined below.

2.1.1 Within One Level

Each node is **responsible for** a specific part of the key space, and it retains that responsibility for as long as it is allocated. A node can meet its space responsibility in two ways. It can **directly contain** entries (data or index terms) for the space. Alternatively, it can **delegate** responsibility for part of the space to a **sibling node**.

A node delegates space to a new sibling node during a node split. A **sibling term** describes a key space for which a sibling node is responsible and includes a **side pointer** to the sibling. A node containing a sibling term is called the **containing node** and the sibling node to which it refers is called the **contained node**.

Any node except the root can contain sibling terms to contained nodes. A **level** of the Π-tree is a maximal connected subgraph of nodes and side pointer edges. Each level of the Π-tree is responsible for the entire key space. The first node at each level is responsible for the whole space, i.e. it is the containing node for the whole key space. Each level describes a partition of the space into subspaces directly contained by nodes of that level. This gives the Π-tree its name.

2.1.2 Multiple Levels

The Π-tree is split from the bottom, like the B-tree. **Data nodes** (leaves) are at level 0. Data nodes contain only data records and/or sibling terms. As the Π-tree grows in height via splitting of a root, new levels are formed.

A split is normally described by an index term. Each **index term**, when posted, includes a **child pointer** to a **child node** and a description of a key space for which the child node is responsible. A node containing the index term for a child node is called a **parent** node. In Π-trees, as in B^{link}-trees, parent nodes are **index nodes** which contain only index terms and/or sibling terms. Parents nodes are at a level one higher than their children.

2.1.3 Well-formed Π-trees

Side pointers and child pointers must refer to nodes which are responsible for spaces that contain the indicated subspaces. A pointer can never refer to a de-allocated node. Further, an index node must contain index terms that refer to nodes that are responsible for spaces, the union of which contains the subspace directly contained by the index node. However, each node at a level need not have a parent node at the next higher level. This is an abstraction and generalization of the idea introduced in the B^{link}-tree[6]. That is, having a new node connected in the B^{link}-tree only via a side pointer is acceptable.

Like [18], we define the requirements of a well-formed general search structure. Thus, a Π-tree is **well-formed** if

1. each node is responsible for a subspace of the search space;
2. each sibling term describes a subspace of its containing node for which its referenced node is responsible.
3. each index term describes a subspace of the index node for which its referenced node is responsible;
4. the union of the spaces described by the index terms and the sibling terms contains the space an index node is responsible for;
5. the lowest level nodes are data nodes.
6. a root exists that is responsible for the entire search space.

The well-formedness description above defines a correct search structure. All structure changing atomic actions must preserve this well-formedness. We will need additional constraints on structure changing actions to facilitate node consolidation (deletion).

2.2 Applicability to Various Search Structures

There are a number of search trees which can be described as Π-trees, and hence exploit our concurrency control and recovery method.

2.2.1 B^{link}-trees

In a B^{link}-tree , an index term (respectively sibling term) is represented by a key value and node pointer. It denotes that the child node (respectively sibling node) referenced is responsible for the entire space greater than or equal to the key. The directly contained space of a B^{link}-tree node is the space it is responsible for minus the space it has delegated to its (one) sibling.

2.2.2 The TSB-tree

A TSB-tree [10] provides indexed access to multiple versions of key sequenced records. As a result, it indexes these records both by key and by time, and we can split by either of these attributes. Historical nodes (nodes created by a split in the time dimension) never split again. History sibling pointers permit the current node directly containing a key space to access history nodes that contain the previous versions of records in that space.

In Figure 1, the region covered by a current node after a number of splits is in the lower right hand corner of the key space it started with. A time split produces a new (historical) node with the original node directly containing the more recent time. A history sibling pointer in the current node refers to the history node. The new history node contains a copy of the prior history sibling pointer.

A key split produces a new (current) node with the original node directly containing the lower part of the key space. A key sibling pointer in the current node refers to the new current node containing the higher part of the key space. The new node will contain a copy of the history sibling pointer. It makes the new current node responsible for not merely its current key space, but for the entire history of this key space.

2.2.3 The hB-Tree

In the hB-tree [11], the idea of containing and contained nodes is explicit and they are described with kd-tree fragments. The "External" markers of [11] can be replaced with the addresses of the nodes which were extracted, and a linking network established with the desired properties. In addition, when the split is by a hyperplane, instead of eliminating the root of the local tree in the splitting node, as in [11], one child of the root (say the right child) points to the new sibling containing the contents of the right subtree. This makes the treatment of hyperplane splits consistent with that of other splits. This is illustrated in Figure 2. A complete description and explanation of hB-tree concurrency, node splitting, and node consolidation is given in [3].

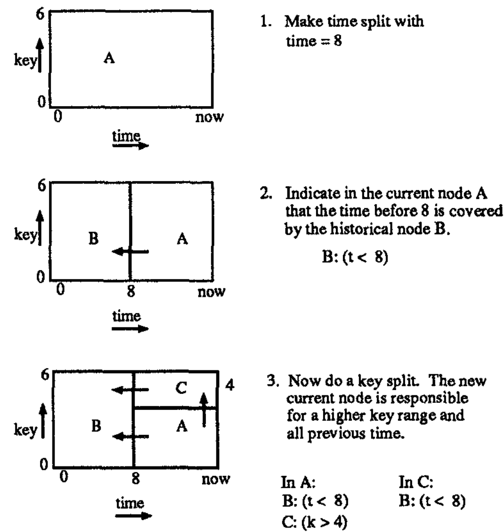

Figure 1: In the Time-Split B-tree, new current nodes contain copies of old history node pointers and old key pointers. New historic nodes contain copies of old history pointers. Current nodes are responsible for all previous time through their historical pointers and all higher key ranges through their key (side) pointers.

3 Π-tree Operations

Here we describe the operations on Π-trees in a very general way. The steps do not describe how to deal with concurrent operations, with failures, or how to decompose structure changes into atomic actions. These are described in later sections.

3.1 Searching

Searches start at the root of the Π-tree. The root is an index node that directly contains the entire search space. In an index node whose directly contained space includes a search point, an index term must exist that references a child node that is responsible for the space that contains the search point. There may be several such child nodes. However, it is desirable to follow the child pointer to the node that directly contains the search point.

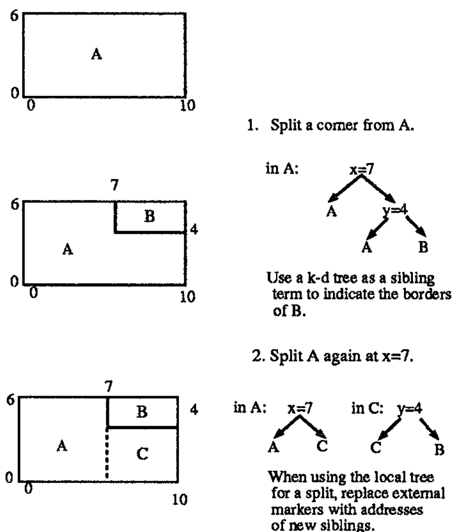

Figure 2: An hB-tree index showing the use of k-d trees for sibling terms. External markers (showing what spaces have been removed in creating "holes") have been replaced with sibling pointers.

This avoids subsequent sibling traversals at the next lower level.

Because the posting of index terms can be delayed, we can only calculate the space **approximately contained** by a child with respect to a given parent. This is the difference between that part of the space of the parent node the child is responsible for and the subspaces that it has delegated to other child nodes referenced by index terms that are **present in the index node**.

We proceed to the child that approximately contains the search point. This node will usually, but not always, contain the search point. If the directly contained space of a node does not include the search point, a side pointer is followed to the sibling node that has been delegated the subspace containing the search point. Eventually, a sibling is found whose directly contained space includes the search point.

The search continues until the data node level of the tree is reached. The record for the search point will be present in the data node whose directly contained space includes the search point, if it exists at all.

3.2 Node Splitting

3.2.1 Node Splitting Steps

A node split has the following steps:

1. Allocate space for the new node.
2. Partition the subspace directly contained by the original node into two parts. The original node continues to directly contain one part. The other part is delegated to the new sibling node.
3. If the node being split is a data node, place in the sibling node all of the original node's data that are contained in the delegated space. Include any sibling terms to subspaces for which the new node is now responsible. Remove from the original node all the data that it no longer directly contains. This partitions the data. (What is dealt with here is point data.)
4. If the node being split is an index node, include in each node the index terms that refer to child nodes whose approximately contained spaces intersect the space directly contained by the node.
5. Put a sibling term in the original node that refers to the new node.
6. Schedule the posting of an index term describing the split to the next higher level of the tree. The index term describes the new sibling and the space for which it is responsible. Posting occurs in a separate atomic action from the action that performs the split.

Example: In a B^{link}-tree, to perform a node split, all records ("records" may be index entries in index nodes or data records in data nodes) whose keys are ordered after the middle record's key are copied from the original node to the new node. The new node has been delegated the high order key subspace. The link (sibling term) is copied from the old node to the new node. Then the copied records are removed from the old node and the link in the old node is replaced with a new sibling term (address of the new node and the split key value).

3.2.2 Clipping

When an index node is split, it is simplest, if possible, to delegate to the new sibling a space which is the union of the approximately contained spaces of a subset of child nodes. However, it can be difficult to split an index node in a multi-attribute index tree in this way because either the space partitioning is too complex, resulting in very large index and sibling terms, or because the division between original and new sibling nodes is too unbalanced, reducing storage utilization. In this case, a child term whose approximately contained space intersects the new

sibling's space as well as the remaining directly contained space in the old sibling is said to be **clipped.** The child term is placed in both parents.

When a child term is clipped, posting index terms describing the subsequent splitting of this child may involve the updating of several parent index nodes. We choose to post to each parent in a separate atomic action. When the split of the child occurs, we post only to the parent that is on the current search path to the splitting node. Other parents can be updated when they are on a search path that results in a sibling traversal to the new node. This exploits a mechanism that is already present to cope with system failures in the midst of Π-tree structure changes. Subsequently, when we refer to "the parent", we intend this to denote the parent that is on the current search path.

3.3 Node Consolidation

When a node becomes under-utilized, it may be possible to consolidate it with either its containing node or one of its contained nodes. We always move the node contents from contained node to containing node, regardless of which is the underutilized node. Then the index term for the contained node is deleted and the contained node is de-allocated. For this to work well,

- both container and contained node must be referenced by index terms in the same parent node, and
- the contained node must only be referenced by this parent.

These conditions mean that only the single parent of the contained node need be updated during a consolidation. This node will also be a parent of the container. This is important as a node cannot be deleted until all references to it have been purged. (This complication arises only in multiattribute Π-trees; B^{link}-tree nodes never have multiple parents.)

There is a difficulty with the above constraints. Whether a node is referenced by more than one parent is not derivable from the index term information we have described thus far. However, multi-parent nodes are only formed when (1) an index node (the parent) splits, clipping one or more of its index terms, or (2) when a child with more than one parent is split, possibly requiring posting in more than one place. We mark these clipped index terms as referring to multi-parent nodes.

4 Atomic Actions for Update

We must ensure that interactions between atomic actions do not cause undetected deadlocks or incorrect searches. Atomic actions must have the all or nothing property and must leave the tree well-formed. All update atomic actions must be correctly serialized. How this is done is described in this section.

4.1 Latching for Atomic Actions

4.1.1 Latch Deadlock Avoidance

Latches can be used for concurrency control involving atomic actions that change an index tree above the leaf level. Latches are semaphores for which the holder's usage pattern guarantees the absence of deadlock. Latches normally come in two modes, **share (S)** mode which permits others with S latches to access the latched data simultaneously, and **exclusive (X)** mode which prevents all other access so that update can be performed. Latches do not involve the database lock manager and latches do not conflict with standard database locks.

Deadlock is avoided by PREVENTING cycles in a "potential delay" graph [8]. If resources are ordered and latched in that order, the potential delay graph can be kept cycle-free without materializing it. Since a Π-tree is usually accessed in search order, we can order parent nodes prior to their children and containing nodes prior to the contained nodes referenced via their side pointers. Space management information can be ordered last.

When arbitrary atomic actions are possible, two phase locking is used to ensure serializability [2]. However, when dealing with index trees, latches can sometimes be released early without violating correctness. This occurs in traversing tree nodes during a search.

Promoting a previously acquired latch violates the ordering of resources and compromises deadlock avoidance. Promotion is the most common cause of deadlock [5]. For example, when two transactions set S latches on the same object to be updated, and then subsequently desire to promote their latches to X, a deadlock results.

Update (U) mode [4] supports promotion. It allows sharing by readers, but conflicts with X and other U modes. An atomic action is not allowed to promote from a S to an X latch. But it may promote from U to X mode. However, a U latch may only be safely promoted to X under restricted circumstances. The rule that we observe is that the promotion request is not made while the requester

holds latches on higher ordered resources. Whenever a node might be written, a U latch is used.

4.1.2 Avoiding Latch-Lock Deadlocks

There are two situations where an index tree atomic action may interact with database transactions and also require locks. These are (1) normal accessing of a database record for fetch, insert, delete, or update and (2) moving data records, whether to split or consolidate nodes.

Should holders of database locks be required to wait for latches on data nodes, this wait is not known to the lock manager and can result in an undetected deadlock even though no deadlock involving only latches is possible. To avoid latch-lock deadlocks, we observe the:

- **No Wait Rule [14]:** An action does not wait for database locks while holding a latch that can conflict with a holder of a database lock.

For our Π-tree operations, we must release latches on data nodes whenever we wait for database locks. However, latches on index nodes may be retained. Except for data node consolidation, no atomic action or database transaction **both:** (i) holds database locks; and (ii) uses other than S latches above the data node level. S latches on index nodes never conflict with database transactions, only with index change atomic actions. Except for consolidate, these actions never hold database locks. And consolidate never requests a U-latch on the index node to be updated while it holds database locks. Hence, its holding of this U-latch cannot conflict with another consolidate (or any other action) that holds database locks. Details of the consolidate operation can be found in [12].

4.2 Page-oriented UNDO

4.2.1 Non-commutative Updates

Data node splitting and consolidation require database locks for some (but not all) recovery protocols. For example, if undos of updates on database records must take place on the same page (leaf node) as the original update, (**page-oriented UNDO**) the records cannot be moved until their updating transaction commits or aborts. No updates can be permitted on records moved by uncommitted structure changes, since undoing the move would cause those records to move. Finally, no update can be permitted that makes the undoing of the move impossible. Such updates are those that consume space in the node that is needed in order to consolidate nodes split by a transaction. Only operations (together with their inverses) that commute with the structure change can be permitted.

When a structure change is part of an independent atomic action, the locks needed for the structure change are two phased but only persist for the duration of this action. All node consolidation is like this. Some data node splitting can also be done in an independent atomic action. If a transaction, T, whose update triggers the need for a node split, has not yet updated any record to be moved by the split, the split can be performed in an action independent of and before T. Then, updates that do not commute with the structure change are only blocked during this independent action. Further, of course, the structure change will not be undone if T aborts.

Other data node splits in page-oriented undo systems must be done within an updating database transaction. In this case, the database locks are held to the end of transaction and the structure change must be undone if the transaction aborts.

4.2.2 Move Locks

For page-oriented undo, a **move** lock is required that conflicts with non-commutative updates. The move lock causes the structure change operation to wait until all transactions that are updating records to be moved have completed. Further, it blocks updating transactions from changing records moved until the moving transaction completes. Finally, the move lock keeps updates from consuming space that would prevent the undoing of the move. Since reads do not require undo, concurrent reads can be tolerated. Hence, move locks are compatible with share mode locks.

When data node splitting occurs in a system with page-oriented undo, the move lock must be held to the end of the transaction T that does the splitting. The posting of the index term for splits cannot occur until and unless T commits, so that undo of the split is possible if T aborts. For the same reason, any other transaction which traverses the sibling pointer created by T's split may not post the index term until T commits. Therefore a move lock must be distinguished from a share lock. A transaction encountering a move lock on a sibling traversal does not schedule an index posting.

A move lock can be realized with a set of individual record locks, a page-level lock, a key-range lock, or even a lock on the whole relation. This depends on the implementation specifics. If the move lock is implemented using a lock whose granule is a node size or larger, once granted, no update activity can alter the locking required. This one lock is sufficient.

Should the move lock be realized as a set of record locks, the need to wait for one of these locks means that the latch on the splitting node must be released. This permits changes to the node that can result in the need for additional records to be locked. Since the space involved—one node—is limited, the frequency of this problem should be low. The node is re-latched and examined for changes (records inserted or deleted). The outcomes possible are no change, different locks required, or even that the move is no longer required.

4.3 Providing Atomicity

We want our approach to index tree concurrency and recovery to work with a large number of recovery methods. Thus, we indicate what our approach requires from a recovery method, without specifying exactly how these requirements are satisfied.

4.3.1 Logging

We assume that write-ahead logging (the WAL protocol) is used. The WAL protocol ensures that actions are logged so as to permit their undo, prior to making changes in the stable database.

Our atomic actions are not user visible and do not involve user commitment promises. Atomic actions need only be "relatively" durable. That is, they must be durable prior to the commitment of transactions that use their results. Thus, it is not necessary to force a "commit" record to the log when an atomic action completes. This "commit" record can be written when the next transaction commits, forcing the log. This transaction is the first one that might depend on the results of the atomic action. This optimization assumes that any transaction which might depend on these results uses the same log.

4.3.2 Identifying an Atomic Action

Atomic actions must complete or partial executions must be rolled back. Hence, the recovery manager needs to know about atomic actions. Three possible ways of identifying an atomic action to the recovery manager are as (i) a separate database transaction, (ii) a special system transaction, or (iii) as a "nested top level action" [13]. Our approach works with any of these techniques, or any other that guarantees atomicity.

5 Structure Changes

A Π-tree structure change is decomposed into a sequence of atomic actions. An update or insert of data into a node can result in a node split. This is one atomic action. The posting of index terms correctly describing the split, and refering to old and new nodes, is a second node update (of the parent of the splitting node) and occurs in a different atomic action. Posting an index term can also result in a node split. Thus, node splitting changes a Π-tree one level at a time. A node consolidation makes changes at two levels of the Π-tree, moving data between nodes and consolidating index terms, in a single atomic action. Consolidation of index terms can lead to further node consolidation, escalating tree changes to the next level, like splitting.

5.1 Completing Structure Changes

There is a window between the time a node splits in one atomic action and the index term describing it is posted in another. Between these atomic actions, a Π-tree is said to be in an intermediate state. We try to complete structure changes because intermediate states may result in non-optimal search or storage utilization.

There are at least two reasons why we "lose track" of which structure changes need completion.

1. A system crash may interrupt a structure change after some of its atomic actions have been executed, but not all.

2. We only schedule the posting of an index term to a single parent.

Structure changes are detected as being incomplete by a tree traversal that includes following a side pointer. At this time, we schedule an atomic action to post the index term. Several tree traversals may follow the same side pointer, and hence try to post the index term multiple times. A subsequent node consolidation may have removed the need to post the index term. These are acceptable because the state of the tree is **testable**. Before posting the index term, we test that the posting has not already been done and still needs to be done.

Node consolidations are scheduled when encountering underutilized nodes. As with node splitting, the Π-tree state is tested to make sure that the consolidation is only performed once, and only when appropriate.

5.2 Exploiting Saved State

Exploiting saved information is an important aspect of efficient index tree structure changes. The bad news of independence is that information about the Π-tree acquired by early atomic actions of the structure change may have changed and so cannot

be trusted by later atomic actions. The Π-tree may have been altered in the interim. Thus, saved information needs to be verified before it is used.

The information that we save consists of search key, nodes traversed on the path from root to data node containing the search key, and the location of the relevant index terms within those nodes. This information can permit us to locate nodes to be restructured without a second search of the nodes on the path, and to find a location within an index node where a new index term is to be inserted or an old one deleted.

To verify saved information, we use state identifiers [9] within nodes to indicate the states of each node. We record these identifiers as part of our saved path. The basic idea is that if a node and its state id (stored in the node) equal saved node and state id, then there have not been any updates to the saved node since the previous traversal. (Log sequence numbers are used for state identifiers in many commercial systems.)

Whether node consolidation is possible has a major impact on how we handle this information. The possibility of removal of a node from the structure affects the extent to which saved information can be trusted.

5.2.1 No Consolidate Case

Consolidation Not Supported [CNS] Invariant: *A node, once responsible for a key subspace, is always responsible for the subspace.* CNS has three effects on our tree operations.

1. During a tree traversal, an index node is searched for an index or sibling term for the pointer to the next node to be searched. We need not hold latches so as to ensure the pointer's continued validity. The latch on an index node can be released after a search and prior to latching a child or sibling node. Only one latch at a time is held during a traversal.

2. When posting an index term in a parent node, it is not necessary to verify the existence of the nodes resulting from the split. These nodes are immortal and remain responsible for the key space assigned to them during the split.

3. During a node split, the parent index node to be updated is either the one remembered from the original traversal (the usual case) or a node that can be reached by following sibling pointers. Thus "re-traversals" to find a parent always start with the remembered parent.

5.2.2 Consolidate Case

Consolidation Possible [CP] Invariant: *A node, once responsible for a key subspace, remains responsible for the subspace only until it is de-allocated.*

De-allocated nodes are not responsible for any key subspace. When re-allocated, they may be used in any way, including being assigned responsibility for different key subspaces, or being used in other indexes. This affects the "validity" of remembered state. Saved path information needs to be verified before being trusted.

The effect CP has on the tree operations is as follows:

1. During a tree traversal, latch coupling is used to ensure that a node referenced via a pointer is not freed before the pointer de-referencing is completed. The latch on the referenced node is acquired prior to the release of the latch on the referencing node. Thus, two latches need to be held simultaneously during a traversal.

2. When posting an index term in a parent node, we must verify that the node produced by the split continues to exist. Thus, in the atomic operation that posts the index term, we also verify that the node that it describes exists by continuing our traversal down to this node.

3. During a node split, the remembered parent node to be updated may have been de-allocated. How to deal with this contingency depends upon how node de-allocation is treated. There are two strategies for handling node de-allocation.

    (a) **De-allocation is NOT a Node Update:** A node's state identifier is unchanged by de-allocation. It is impossible to determine by state identifier examination if a node has been de-allocated. However, we ensure that the root does not move and is never de-allocated. Then, any node reachable from the root via a tree traversal is guaranteed to be allocated. Thus, tree re-traversals start at the root. A node on the path is accessed and latched using latch coupling, just as in the original traversal. Typically, a path re-traversal is limited to re-latching path nodes and comparing new state ids with remembered state ids, which will usually be equal.

    (b) **De-allocation is a Node Update:** Node de-allocation changes not only space

management information, but also the node's state identifier to indicate that de-allocation has taken place. This requires the posting of a log record and possibly an additional disk access. However, the remembered parent node in the path will always be allocated if its state identifier has not changed and re-traversals can begin from there. If it has changed, however, one must go up the path, setting and releasing latches until a node with an unchanged state id is found or the root is encountered. A path re-traversal begins at this node. Since node de-allocation is rare, full re-traversals of the tree are usually avoided.

5.3 A Structure Change Action

To illustrate the detailed steps of an atomic action, we treat the case of posting index terms in a B^{link}-tree where node consolidation is possible and de-allocation is not a node update. The arguments for the index term posting operation are: The LEVEL of the tree where the index term will be posted, the remembered PATH, the ADDRESS of the new node, and the KEY which was searched for.

Index term posting for the B^{link}-tree performs the following steps:

1. **Search:** The search starts from the root using latch-coupling with S-latches. As long as the state identifiers are unchanged, the remembered PATH is used. If a state identifier is changed, a search for the KEY begins. When the LEVEL is reached, U-latches are used, possibly traversing side pointers until the correct NODE is U-latched. The parent of NODE is left S-latched.

2. **Verify Split:**. If the index term has already been posted, the action is terminated. Otherwise the child node with the largest index term key value smaller than the KEY is S latched. It is accessed to determine whether a side pointer refers to a sibling node that is responsible for the space that contains the KEY. If not, then the node whose index term is being posted has already been deleted and the action is terminated.

   If a sibling exists that is responsible for space containing the KEY, this sibling becomes the one whose index term is posted. (This may mean a new ADDRESS will be in the new index term.) The S latches are dropped. The U latch on NODE is promoted to an X latch. (The new node whose index term is being posted cannot be consolidated while a latch is held on NODE.)

3. **Space Test:** Test NODE for sufficient space to accommodate the update. If sufficient, proceed to **Update Node**.

   Otherwise, split NODE: The space management information is X latched and a new node is allocated. The key space directly contained by the current node is divided, such that the new node becomes responsible for a subspace of the key space. A sibling term that references the new node is placed in NODE. The change to NODE and the creation of the new node are logged. If NODE is not the root, an index term is generated containing the new node's address as a child pointer. (At the end of the action, when all latches are released, an index posting operation is scheduled for the parent of NODE.)

   If NODE is the root, a second node is allocated. NODE's contents are removed from the root and put into this new node. A pair of index terms is generated that describe the two new nodes and they are posted to the root. These changes are logged.

   Then check which resulting node has a directly contained space that includes KEY, and make that NODE. This can require descending one more level in the Π-tree should NODE have been the root where the split causes the tree to increase in height. Release the X latch on the other node, but retain the X latch on NODE. Repeat this **Space Test** step.

4. **Update NODE:** Post the index term in NODE. Post a log record describing the update to the log. Release all latches still held by the action.

6 Discussion

There are many ways to realize Π-tree updates and associated structure changes. We describe a specific complete procedure in [12]. In this paper, we have emphasized the abstract elements of our approach. These elements include

- extending the notion of sibling link to encompass a much wider class of tree-like structures which we have called Π-trees;
- decomposing updates into a number of atomic actions so as to separate structure changes that

alter the index from updates in database transactions;

- completion of structure changes when incomplete structure changes are encountered during normal processing;
- using latches for efficient concurrency control without deadlock;
- dealing with node consolidation as well as its absence.

Our approach to index tree structure changes provides high concurrency while being usable with many recovery schemes and with many varieties of index trees. We have described it in an abstract way which emphasizes its generality and hopefully makes the approach understandable.

Our techniques permit multiple concurrent structure changes. In addition, all update activity and structure change activity above the data level executes in short independent atomic actions which do not impede normal database activity. Only data node splitting might execute in the context of a database transaction, and even here the resulting index term posting is separate from the database transaction. Should the recovery method support non-page-oriented UNDO, even data node splitting can occur outside of the database transaction.

References

[1] Bayer, R., Schkolnick, M., Concurrency of operations on B-trees. *Acta Informatica* 9,1 (1977) 1-21.

[2] Eswaren, K., Gray, J., Lorie, R. and Traiger, I. On the notions of consistency and predicate locks in a database system. *Comm ACM* 19,11 (Nov 1976) 624-633.

[3] Evangelidis, G., Lomet, D. and Salzberg, B., Modifications of the hB-tree for node consolidation and concurrency. (in preparation)

[4] Gray, J.N., Lorie, R. A., Putzulo, G. R., and Traiger, I. L., Granularity of locks and degrees of consistency in a shared data base. *IFIP Working Conf on Modeling of Data Base Management Systems* (1976) 1-29.

[5] Gray, J. and Reuter, A.,*Transaction Processing: Techniques and Concepts*, Morgan Kaufmann (book in preparation).

[6] Lehman, P., Yao, S.B., Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Systems* 16,4 (Dec 1981) 650-670.

[7] Lomet, D. B. Process structuring, synchronization, and recovery using atomic actions. *Proc. ACM Conf. on Language Design for Reliable Software*, SIGPLAN Notices 12,3 (Mar 1977) 128-137.

[8] Lomet, D.B. Subsystems of processes with deadlock avoidance. *IEEE Trans. on Software Engineering* SE-6,3 (May 1980) 297-304.

[9] Lomet, D.B. Recovery for shared disk systems using multiple redo logs. Digital Equipment Corp. Tech ReportCRL90/4 (Oct 1990) Cambridge Research Lab, Cambridge, MA

[10] Lomet, D. and Salzberg, B., Access methods for multiversion data, *Proc. ACM SIGMOD Conf.*, Portland, OR (May 1989) 315-324.

[11] Lomet, D. and Salzberg, B., The hB-tree: a multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Systems* 15,4 (Dec 1990) 625-658.

[12] Lomet, D. and Salzberg, B. Concurrency and Recovery for Index Trees. Digital Equipment Corp. Tech Report CRL91/8 (Aug 1991) Cambridge Research Lab, Cambridge, MA

[13] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, P., and Schwarz, P. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. (to appear in *ACM Trans. Database Systems*).

[14] Mohan, C. and Levine, F., ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. (to appear in *Proc. ACM SIGMOD Conf.*, San Diego, (June 1992)).

[15] Sagiv, Y., Concurrent operations on B* trees with overtaking. *J Computer and System Sciences* 33,2 (1986) 275-296

[16] Salzberg, B., Restructuring the Lehman-Yao tree. *Northeastern University Tech Report* TR BS-85-21 (1985), Boston, MA

[17] Shasha, D., Goodman, N., Concurrent search structure algorithms. *ACM Trans. Database Systems* 13,1 (Mar 1988) 53-90.

[18] Srinivasan, V. and Carey, M. Performance of B-tree concurrency control algorithms. *Proc. ACM SIGMOD Conf.*, Denver, CO (May 1991) 416-425.